Building Program Behavior Models

Mikhail Auguston
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003-0001, USA
phone: (505) 646-5286, fax: (505) 646-1002
e-mail: `mikau@cs.nmsu.edu`

Abstract. This paper suggests an approach to the development of software testing and debugging automation tools based on precise program behavior models. The program behavior model is defined as a set of events (event trace) with two basic binary relations over events -- precedence and inclusion, and represents the temporal relationship between actions. A language for the computations over event traces is developed that provides a basis for assertion checking, debugging queries, execution profiles, and performance measurements.

The approach is nondestructive, since assertion texts are separated from the target program source code and can be maintained independently. Assertions can capture both the dynamic properties of a particular target program and can formalize the general knowledge of typical bugs and debugging strategies. An event grammar provides a sound basis for assertion language implementation via target program automatic instrumentation. Event grammars may be designed for sequential as well as for parallel programs. The approach suggested can be adjusted to a variety of programming languages.

Keywords. Program behavior models, events, event grammars, software testing and debugging

1 Introduction

Dynamic program analysis is one of the least understood activities in software development. A major problem is still the inability to express the mismatch between the expected and the observed behavior of the program on the level of abstraction maintained by the user [11]. In other words, a flexible and expressive specification formalism is needed to describe properties of the software system's implementation. Program testing and debugging is still a human activity performed largely without any adequate tools and consuming more than 50% of the total program development time and effort [10]. Debugging concurrent programs is even more difficult because of parallel activities, non-determinism and time-dependent behavior.

One way to improve the situation is to partially automate the debugging process. Precise *model of program behavior* becomes the first step towards debugging automation. It appears that traditional methods of programming language semantics definition don't address this aspect.

In building such a model several considerations were taken in account. The first assumption we make is that the model is discrete, i.e. comprises a finite number of well-separated elements. This assumption is typical for Computer Science methods used for static and dynamic analysis of programs. For this reason the notion of *event* as an elementary unit of action is an appropriate basis for building the whole model. The event is an abstraction for any detectable action performed during the program execution, such as a statement execution, expression evaluation, procedure call, sending and receiving a message, etc.

Actions (or events) are evolving in time and the program behavior represents the temporal relationship between actions. This implies the necessity to introduce an ordering relation for events. Semantics of parallel programming languages and even some sequential languages (such as C) don't require the total ordering of actions, so *partial event ordering* is the most adequate method for this purpose [17].

Actions performed during the program execution are at different levels of granularity, some of them include other actions, e.g. a subroutine call event contains statement execution events. This consideration brings to our model *inclusion relation*. Under this relationship events can be hierarchical objects and it becomes possible to consider program behavior at appropriate levels of granularity.

Finally, the program execution can be modeled as a set of events (*event trace*) with two basic relations: partial ordering and inclusion. The event trace actually is a model of program's behavior temporal aspect. In order to specify meaningful program behavior properties we have to enrich events with some attributes. An event may have a type and some other attributes, such as event duration, program source code related to the event, program state associated with the event (i.e. program variable values at the beginning and at the end of event), etc.

The next problem to be addressed after the program behavior model is set up is the formalism specifying properties of the program behavior. This could be done in many different ways, e.g. by adopting some kind of logic calculi (predicate logic, temporal logic). Such a direction leads to tools for program static verification, or in more pragmatic incarnations to an approach called model checking [13] As indicated in [1] “Dynamic analysis is limited to checking observed behaviors, and so in principle provides weaker assurances, but this is balanced by checking a wider range of properties and typically by better performance ... .”

Since our goal is debugging automation, i.e. a kind of program dynamic analysis that requires different types of assertion checking, debugging queries, program execution profiles, and so on, we came up with the concept of a *computation over the event trace*. It seems that this concept is general enough to cover all the above mentioned needs in the unifying framework, and provides sufficient flexibility. This approach implies the design of a special programming language for computations over the event traces. We suggest a particular language called FORMAN [2], [4], [15] based on functional paradigm and the use of event patterns and aggregate operations over events.

Patterns describe the structure of events with context conditions. Program paths can be described by path expressions over events. All this makes it possible to write assertions not only about variable values at program points but also about data and control flows in the target program. Assertions can also be used as conditions in rules which describe debugging actions. For example, an error message is a typical action for a debugger or consistency checker. Thus, it is also possible to specify debugging strategies.

The notions of event and event type are powerful abstractions which make it possible to write assertions independent of any target program. Such generic assertions can be collected in standard libraries which represent the general knowledge about typical bugs and debugging strategies and could be designed and distributed as special software tools.

FORMAN is a powerful and general language to describe computations over program event trace that can be considered as an example of *a special programming paradigm*. Possible application areas include program testing and debugging, performance measurement and modeling, program profiling, program animation, program maintenance and program documentation [6]. A study of FORMAN application for parallel programming is presented in [5]

2 Events

FORMAN is based on a semantic model of target program behavior in which the program execution is represented by a set of events. An *event* occurs when some action is performed during the program execution process. For instance, a message is sent or received, a statement is executed, or some expression is evaluated. A particular action may be performed many times, but every execution of an action is denoted by a unique event.

Every event defines a time interval which has a beginning and an end. For atomic events, the beginning and end points of the time interval will be the same. All events used for assertion checking and other computations over event traces must be detectable by some implementation (e.g. by an appropriate target program instrumentation.) Attributes attached to events bring additional information about event context, such as current variable and expression values.

In order to give some support for our notion of event let us consider a well-known idea such as a counter. Usually the history of a variable X when used as a counter looks like:

```
X := 0; ...
Loop ...
        X := X + 1; ...
endloop; ...
```

In order to check whether the actual behavior of the counter X matches the pattern described by the program fragment above we have to consider the following events. Let Initialize_X denote the event of assigning 0 to the variable X, Augment_X denote the event of incrementing X, and Assign_X denote an event of assigning any value to the variable X. The event of the type Assign_X is a composite one; it contains either Initialize_X or Augment_X type events. One could check whether X behaves as a counter when a program segment S is executed in the following way. First, the sequence A of all events of the type Assign_X from the event trace of program segment S has to be extracted preserving the ordering between events. Second, A has to be matched with the pattern:

```
Initialize_X    (Augment_X) *
```

where’*’ denotes repetition zero or more times. If the actual sequence of events does not match this pattern we can report an error. Therefore, assertion checking can be represented as a kind of computation over target program event trace.

Another informal example involves parallel events. Let us suppose that Assign_Y denotes an event of assigning a value to the shared variable Y through any of several parallel processes. Then, detecting a set of events of the type Assign_Y that happen “at the same time” (i.e. are not under the precedence relation) may be evidence of a possible data-race condition in the program execution.

The program state (current values of variables) can be considered at the beginning or at the end of an appropriate event. This provides the opportunity to write assertions about program variable values at different points in the program execution history.

Program profiling usually is based on counting the number of events of some type, e.g. the number of statement executions or procedure calls. Performance measurements may be based on attaching the duration attribute to such events and summarizing durations of selected events.

3 The Language for Computations Over Event Traces

FORMAN is a high-level specification language for expressing intended behavior or known types of error conditions when debugging or testing programs. It is intended to be used in conjunction with a high-level programming language which is called the *target language*.

The model of target program behavior is formally defined through a set of general axioms about two basic relations, which may or may not hold between two arbitrary events: they may be sequentially ordered (PRECEDES), or one of them might be included in another composite event (IN). For each pair of events in the event trace no more than one of these relations can be established.

There are several general axioms that should be satisfied by any events a, b, c in the event trace of any target program.

1) Mutual exclusion of relations.

```
a PRECEDES b  => not (a IN b)
a IN b  =>  not(a PRECEDES b)
```

2) Noncommutativity.

```
a PRECEDES b  =>  not( b PRECEDES a)
a IN b  =>  not( b IN a)
```

3) Transitivity.

```
(a PRECEDES b ) and ( b PRECEDES c ) =>
             ( a PRECEDES c)
```

Irreflexivity for PRECEDES and `IN` follows from 2). Note that PRECEDES is an irreflexive partial ordering.

4) Distributivity

```
(a IN b) and (b PRECEDES c) =>
       (a PRECEDES c)

(a PRECEDES b) and (c IN b) =>
       (a PRECEDES c)

(FOR ALL a IN b
       (FOR ALL c IN d (a PRECEDES c) ))
             =>     (b PRECEDES d)
```

In order to define the behavior model for some target language, types of events are introduced. Each event belongs to one or more of predefined event types, which are induced by target language abstract syntax (e.g. execute-statement, send-message, receive-message) or by target language semantics (rendezvous, wait, put-message-in-queue).

The target program execution model is defined by an event grammar. The event may be a compound object and the grammar describes how the event is split into other event sequences or sets. For example, the event execute-assignment-statement contains a sequence of events evaluate-right-hand-part and execute-destination. The evaluate-right-hand-part, in turn, consists of an unique event evaluate-expression. The event grammar is a set of axioms that describe possible patterns of basic relations between events of different type in the program execution history, it is not intended to be used for parsing actual event trace.

The rule `A :: ( B C)` establishes that if an event `a` of the type A occurs in the trace of a program, it is necessary that events b and c of types B and C, also exist, such that the relations `b IN a, c IN a, b PRECEDES c` hold.

For example, the event grammar describing the semantics of a PASCAL subset may contain the following rules. The names, such as `execute-program,` and `ex-stmt` in the grammar denote event types.

```
execute-program  :: ( ex-stmt * )
```

This means that each event of the type `execute-program` contains an ordered (w.r.t. relation PRECEDES) sequence of zero or more events of the type `ex-stmt`.

```
ex-stmt :: (  label? ( ex-assignment |
     ex-read-stmt | ex-write-stmt |
     ex-reset-stmt | ex-rewrite-stmt |
     ex-close-stmt | ex-cond-stmt |
     ex-loop-stmt | call-procedure) )
```

The event of the type `ex-stmt` contains one of the events `ex-assignment, ex-read-stmt,` and so on. This inner event determines the particular type of statement executed and may be preceded by an optional event of the type `label` (traversing a label attached to the statement).

```
ex-assignment  ::
     (ex-righthand-part  destination)
```

The order of event occurrences reflects the semantics of the target language. When performing assignment statement first the right-hand part is evaluated and after this the destination event occurs (which denotes the assignment event itself). The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events.

An event has attributes, for instance, source text fragment from the corresponding target program, current values of target program variables and expressions at the beginning and at the end of event, duration of the event, previous path (i.e. set of events preceding the event in the target program execution history), etc.

FORMAN supplies a means for writing assertions about events and event sequences and sets. These include quantifiers and other aggregate operations over events, e.g., sequence, bag and set constructors, boolean operations and operations of target language to write assertions on target program variables.

Events can be described by patterns which capture the structure of event and context conditions. Program paths can be described by regular path expressions over events.

The main extension for the parallel case consists of the introduction of a new kind of composite event -- “snapshot,” which can be considered as an abstraction for the notion “a set of events that may happen at the same time.” The “snapshot” event makes it possible to describe and to detect at run-time such typical parallel processing faults as data races and deadlock states.

All this makes it possible to formalize assertions of the following types:

- “all variables in the program must be initialized before using in some expression,”
- “file must be opened, then the read statement is performed zero or more times and after that the close statement is executed,”
- “at least one variable changes its value during one loop L iteration,”
- “after the execution of a subprogram P the value of variable X remains unchanged,”
- “there is an attempt to assign values to the same variable in two parallel processes” (data race condition),
- “deadlock for parallel processes P1 and P2 is detected.”

In addition to debugging and testing, FORMAN can also be used to specify profiles and performance measurements.

4 Examples of Debugging Rules and Queries

In general, a *debugging rule* performs some actions that may include computations over the target program execution history. The aim is to generate informative messages and to provide the user with some values obtained from the trace in order to detect and localize bugs. Rules can provide dialog to the user as well. An assertion is a boolean expression that may contain quantifiers and sequencing constraints over events.

Assertions can be used as conditions in the rules describing actions that can be performed if an assertion is satisfied or violated. A debugging rule has the form:

assertion **SAY** (expression sequence)

ONFAIL SAY (expression sequence)

The presence of metavariables in the assertion makes it possible to use FORMAN as a debugger query language. The computation of an assertion is interrupted when it becomes clear that the final value will be False, and the current values of metavariables can be used to generate readable and informative messages.

The following examples have been executed on our prototype FORMAN/PASCAL assertion checker [3], [4]. The PASCAL program reads a sequence of integers from file XX.TXT.

```
program e1;
     var X: integer;
     XX: file of text;
begin
     X:= 7;
     (* initial value is assigned here *)
     reset (XX, ‘XX.TXT’);
     while X<>0 do
            read(XX, X)
end.
```

The contents of the file XX.TXT are as follows:

```
11 5 3 7 8 9 3 13 2 3 45 8 754 45567 0
```

Query 1. In order to obtain the history of variable X the following computation over event trace can be performed. The rule condition is TRUE, and is shown as a side effect the whole history of variable X.

```
TRUE
SAY ( 'The history of variable X is:'
[D: destination  IS X FROM execute_program
  APPLY VALUE(D) ]  )
```

The `[ ... ]` construct above defines a loop over the whole program execution trace (`execute_program` event). All events matching the pattern `destination IS X` are selected from the trace and the function VALUE is applied to them. The resulting sequence consists of values assigned to the X variable during the program execution.

When executed on our prototype the following output is produced:

```
Assertion #1 checked successfully...
The history of variable X is: 7 11 5 3 7 8
9 3 13 2 45 8 754 45567 0
```

Assertion 2. Let’s write and check the assertion : *“The value of* variable X does not exceed 17."

```
FOREACH *S: ex_stmt
     CONTAINS (D: destination IS X)
     FROM execute_program
            VALUE(D) < 17
ONFAIL
SAY('Value ' VALUE(D)
'is assigned to the variable X in stmt ')
SAY(S) SAY('This is record #'
CARD[ ex_read_stmt FROM PREV_PATH(S)] + 1
'in the file XX.TXT')
```

We check the assertion for all events where the value of X may be altered. These are events of the type `destination` which can appear within `ex_assignment_stmt` or `ex_read_stmt` events. In order to make error messages about assertion violations more informative we include the embracing event of the type `ex_stmt`. Metavariables `S` and `D` refer to those events of interest. When the assertion is violated for the first time, the assertion evaluation terminates and current values of metavariables can be used for message output. The value of a metavariable when printed by the SAY clause is shown in the form:

```
event-type:> event-source-text
Time= event-begin-time .. event-end-time
```

Event begin and end times in this prototype implementation are simply values of step counter.

Since we expect the assertion might be violated when executing a Read statement, it makes sense to report the record number of the input file `xx.txt` where the assertion is violated. The program state does not contain any variables which values could provide this information. But we can perform auxiliary calculations independently from the target program using FORMAN aggregate operations. In this particular case the number of events of the type `ex_read_stmt` preceding the interruption moment is counted. This number plus 1 (since the violation occurs when the read statement is executed) yields the number of an input record on which the variable X was first assigned the value exceeding 17.

```
Assertion # 2 violation!
 Value 45 is assigned to the variable X in stmt
ex_stmt :> Read( XX , X )   Time= 73 .. 78
This is record # 11 in the file XX.TXT
```

Query 3. Profile measurement. In order to obtain the actual number of statements executed, the following query can be performed:

```
TRUE
SAY('The total number of statements executed is:'
CARD[ ALL ex_stmt FROM execute_program ])
```

The `ALL` option in the aggregate operation indicates that all nested events of the type `ex_stmt` should be taken into account.

```
 Assertion #3 checked successfully...
The total number of statements executed is: 18
```

Example of a *generic assertion* which must be true for any program in the target language.

"Each variable has to be assigned value before it is used in an expression evaluation."

```
FOREACH * S: ex_stmt FROM execute_program
     FOREACH * E: eval_expression
            CONTAINS (V: variable) FROM S
EXISTS D: destination FROM PREV_PATH(E)
     SOURCE_TEXT(D) = SOURCE_TEXT(V)
ONFAIL SAY( 'In event' S)
     SAY( 'in expression evaluation')
     SAY(E)
     SAY('uninitialized variable'
            SOURCE_TEXT(V) 'is used')
```

For the following PASCAL program our prototype detects the presence of the bug described above.

```
program e2;
var X,Y: integer;
begin      Y:= 3;
     if Y < 2 then begin
            X:= 7; Y:= Y + X
     else Y:= X - Y
(*** here the error appears:
     X has no value! ***)
end.
```

```
Assertion #4 violation!
In event ex_stmt :> If ( Y < 2 ) then X :=
7 ; Y := ( Y + X ) ; else Y := ( X - Y ) ;
Time= 10 .. 35
  in expression evaluation
  eval_expression :> ( X - Y ) Time= 20 .. 29
  uninitialised variable X is used
```

Debugging rules can be considered as a way of formalizing reasoning about the target program execution -- humans often use similar patterns for reasoning when debugging programs. For example, if the index expression of an array element is out of the range, the debugger can try a rule for eval-index events that invokes another rule about wrong value of the event eval-expression, which in turn will cause investigation of histories of all variables included in the expression.

Yet another application of generic assertions and debugging rules may be for describing run-time constraints (sequences of procedure calls, actual parameter dependences, etc.) for nontrivial subroutine packages, e.g. for the MOTIF package for GUI design. A library containing assertions and debugging rules relevant to such a package may be useful for writing C programs calling subroutines from the package.

5 Related Work

What follows is a very brief survey of basic ideas known in Debugging Automation to provide the background for the approach advocated in this paper. More references to related work and a detailed survey may be found in [4]

5.1 Event Notion

The Event Based Behavioral Abstraction (EBBA) method suggested in [8] characterizes the behavior of the whole program in terms of both primitive and composite events. Context conditions involving event attribute values can be used to distinguish events. EBBA defines two higher level means for modeling system behavior -- clustering and filtering. Clustering is used to express behavior as composite events, i.e. aggregates of previously defined events. Filtering serves to eliminate from consideration events which are not relevant to the model being investigated. Both event recognition and filtering can be performed at run-time.

An event-based debugger for the C programming language called Dalek [22] provides a means for description of user-defined events which typically are points within a program execution trace. A target program has to be instrumented in order to collect values of event attributes. Composite events can be recognized at run-time as collections of primitive events.

FORMAN has a more comprehensive modelling approach than EBBA or Dalek, based on the event grammar. A language for expressing computations over execution histories is provided, which is missing in EBBA and Dalek. The event grammar makes FORMAN suitable for automatic source code instrumentation to detect all necessary events. FORMAN supports design of universal assertions and debugging rules that could be used for debugging of arbitrary target programs. This generality is missing in EBBA and Dalek approaches. The event in FORMAN is a time interval, in contrast with the event notion in previous approaches where events are considered pointwise time moments.

5.2 Path Expressions

Data and control flow descriptions of the target program are essential for testing and debugging purposes. It is useful to give such a description in an explicit and precise form. The path expression technique introduced for specifying parallel programs in [12] is one such formalism. Trace specifications also are used in [21] for software specification. This technique has been used in several projects as a background for high-level debugging tools, (e.g. in [11]), where path rules are suggested as kinds of debugger commands. FORMAN provides flexible language means for trace specification including event patterns and regular expressions over them.

5.3 Assertion Languages

Assertion (or annotation) languages provide yet another approach to debugging automation. The approaches currently in use are mostly based on boolean expressions attached to selected points of the target program, like the assert macro in C. The ANNA [19] annotation language for the Ada target language supports assertions on variable and type declarations. In the TSL [18], [24] annotation language for Ada the notion of event is introduced in order to describe the behavior of Tasks. Patterns can be written which involve parameter values of Task entry calls. Assertions are written in Ada itself, using a number of special pre-defined predicates. Assertion-checking is dynamic at run-time, and does not need post-mortem analysis. The RAPIDE project [20] provides a reach event-based assertion language for software architecture description.

In [7] events are introduced to describe process communication, termination, and connection and detachment of process to channels. A language of Behavior Expressions (BE) is provided to write assertions about sequences of process interactions. BE is able to describe allowed sequences of events as well as some predicates defined on the values of the variables of processes. Event types are process communication and interactions such as send, receive, terminate, connect, detach. Evaluation of assertions are done at run-time. No composite events are provided.

Another recent experimental debugging tool is based on trace analysis with respect to assertions in temporal interval logic. This work is presented in [16] where four types of events are introduced: assignment to variables, reaching a label, interprocess communication and process instantiation or termination. Composite events cannot be defined. Different varieties of temporal logic languages are used for program static analysis called Model Checking [13].

In [25] a practical approach to programming with assertions for the C language is advocated, and it is demonstrated that even local assertions associated with particular points within the program may be extremely useful for program debugging.

The FORMAN language for computations over traces provides flexible means for writing both local and global assertions, including those about temporal relations between events.

5.4 Algorithmic Debugging

The original algorithmic program debugging method was introduced in [27] for the Prolog language. In [26] and [14] this paradigm is applied to a subset of PASCAL.

The debugger executes the program and builds a trace execution tree at the procedure level while saving some useful trace information such as procedure names and input/output parameter values. The algorithmic debugger traverses the execution tree and interacts with the user by asking about the intended behavior of each procedure. The user has the possibility to answer “yes” or “no” about the intended behavior of the procedure. The search finally ends and a bug is localized within a procedure *p* when one of the following holds: procedure *p* contains no procedure calls, or all procedure calls performed from the body of procedure *p* fulfill the user’s expectations.

Algorithmic debugging can be considered as an example of debugging strategy, based on some assertion language (in this case assertions about results of a procedure call.) The notion of computation over execution trace introduced in FORMAN may be a convenient basis for describing such debugging strategies.

6 Conclusions

In brief, our approach can be explained as “computations over a target program event trace.” We expect the advantages of our approach to be the following:

- The notion of **an event grammar** provides a general basis for program behavior models. In contrast with previous approaches, the **event** is not a point in the trace but an interval with a beginning and an end.
- Event grammar provides a coordinate system to refer to any interesting event in the execution history. Program variable values are attributes of an event’s beginning and end. Event attributes provide complete **access to each target program’s execution state**. Assertions about particular execution states as well as assertions about sets of different execution states may be checked.
- The PRECEDES relation yields a **partial order** on the set of events, which is a natural model for parallel program behavior.
- The IN relation yields a **hierarchy of events**, so the assertions can be defined at an appropriate level of granularity.
- A language for **computations over event traces** provides a **uniform framework** for assertion checking, profiles, debugging queries, and performance measurements.
- The access to the complete target program execution history and the ability to formalize **generic assertions** can be used in order to define **debugging rules and strategies.**
- The fact that assertions and other computations over target program event trace can be **separated from the text of the target program** allows accumulation of formalized knowledge about particular programs and about the whole target language in separate files. This makes it easy to control the amount of assertions to be checked.

According to [9] and [23] approximately 40-50% of all bugs detected during the program testing are logic, structural, and functionality bugs, i.e. bugs which could be detected by appropriate assertion checking similar to the demonstrated above.

It appears that the approach initially designed for program behavior modeling may be used in other dynamic system behavior models as well. The methodology is based on identifying event types representing essential actions performed within the system, and defining the basic relations PRECEDES and IN for those events (event grammar), and appropriate event attributes. Then the FORMAN-like language for computations over event traces may be developed to specify behavior properties, to perform queries and other kinds of dynamic analysis.

References

[1] F. Anger, R. Rodriguez, M. Young, “Combining Static and Dynamic Analysis of Concurrent Programs”, *Proceedings of International IEEE Conference on Software Maintenance*, Victoria, BC, Canada, Sept. 1994, pp.89-98.

[2] M. Auguston, “FORMAN -- A Program Formal Annotation Language”, *Proceedings of the 5:th Israel Conference on Computer Systems and Software Engineering*, Gerclia, May 1991, IEEE Computer Society Press, 149-154.

[3] M. Auguston, “A language for debugging automation”, *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering*, Jurmala, June 1994, Knowledge Systems Institute, pp. 108-115.

[4] M. Auguston, “Program Behavior Model Based on Event Grammar and its Application for Debugging Automation”, *in Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, France, May 1995.

[5] M. Auguston, P. Fritzson, “PARFORMAN -- an Assertion Language for Specifying Behavior when Debugging Parallel Applications”, *International Journal of Software Engineering and Knowledge Engineering*, vol.6, No 4, 1996, pp. 609-640.

[6] M. Auguston, A. Gates, M. Lujan, “Defining a program Behavior Model for Dynamic Analyzers”, *Proceedings of the 9th*

International Conference on Software Engineering and Knowledge Engineering, SEKE'97, Madrid, Spain, June 1997, pp. 257-262

[7] F. Baiardi, N. De Francesco, G. Vaglini, "Development of a Debugger for a Concurrent Language", *IEEE Transactions on Software Engineering*, vol. SE-12, No. 4, April 1986, pp. 547-553.

[8] P. C. Bates, J. C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", *The Journal of Systems and Software* 3, 1983, pp. 255-264.

[9] B. Beizer, Software Testing Techniques, Second Edition, International Thomson Computer Press, 1990.

[10] F. Brooks, The Mythical Man-Month, 2nd edition, Addison-Wesley, 1995.

[11] B. Bruegge, P. Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", *The Journal of Systems and Software* 3, 1983, pp. 265-276.

[12] R. H. Campbell, A. N. Habermann, "The specification of process synchronization by path expressions", *Lecture Notes in Computer Science*, vol. 16, 1974, pp. 89-102.

[13] E.Clarke et al., "Verification tools for Finite State Concurrent Systems", LNCS vol.803, 1994, pp.124-175.

[14] P. Fritzson, N. Shahmehri, M. Kamkar, T. Gyimothy, "Generalized Algorithmic Debugging and Testing", *ACM LOPLAS -- Letters of Programming Languages and Systems*. Vol. 1, No. 4, December 1992.

[15] P. Fritzson, M. Auguston, N. Shahmehri, "Using Assertions in Declarative and Operational Models for Automated Debugging", *The Journal of Systems and Software* 25, 1994, pp. 223-239.

[16] G. Goldszmidt, S. Katz, S. Yemini, "Interactive Blackbox Debugging for Concurrent Languages", *SIGPLAN Notices* vol. 24, No. 1, 1989, pp. 271-282.

[17] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, No. 7, July 1978, pp. 558-565.

[18] D. C. Luckham, D. Bryan, W. Mann, S. Meldal, D. P. Helmbold, "An Introduction to Task Sequencing Language, TSL version 1.5" (Preliminary version), Stanford University, February 1, 1990, pp. 1-68.

[19] D. C. Luckham, S. Sankar, S. Takahashi, "Two-Dimensional Pinpointing: Debugging with Formal Specifications", *IEEE Software*, January 1991, pp.74-84.

[20] D. Luckham, J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering, Vol.21, No. 9, 1995, pp. 717-734.*

[21] J. McLean, "A Formal Method for the Abstract Specification of Software", *Journal of the Association of Computing Machinery*, vol.31, No. 3, July 1984, pp. 600-627.

[22] R. Olsson, R. Crawford, W. Wilson, "A Dataflow Approach to Event-based Debugging", *Software -- Practice and Experience, Vol.21(2), February 1991, pp. 19-31.*

[23] S. L. Pfleeger, Software Engineering, Theory and Practice, Prentice hall, 1998.

[24] D. Rosenblum, "Specifying Concurrent Systems with TSL", *IEEE Software*, May 1991, pp.52-61.

[25] D. Rosenblum, "A Practical Approach to Programming With Assertions", *IEEE Transactions on Software Engineering, Vol. 21, No 1, January 1995, pp. 19-31.*

[26] N. Shahmehri, "Generalized Algorithmic Debugging", Ph.D. Thesis No. 260, Dept. of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden, 1991.

[27] E. Shapiro, "Algorithmic Program Debugging", MIT Press, May 1982.